KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 1 ⋄ poznámky k přednášce

11. Polynomy a stromy

Přednáška tento týden neproběhla. Náhradou jsou dva úkoly na samostudium: polynomy a stromy. Upozorňuji, že kvůli časové tísni jsem kód dostatečně netestoval, takže v něm mohou být drobné chyby.

1 Polynomy

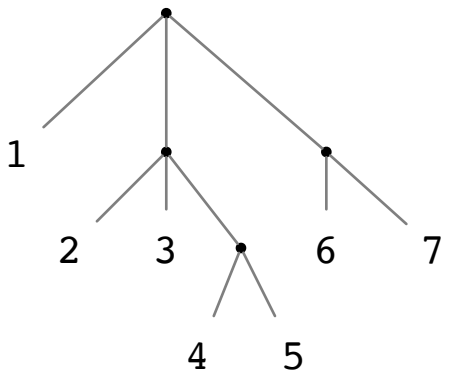
Tuto část si nastudujte z textu Programy a projekty v jazyku Scheme III, kap. 2.6 (str. 40–47).

2 Jednoduché stromy

Viděli jsme, že páry lze použít k reprezentaci datových struktur s dvěma složkami a, pokud použijeme párů více, i k reprezentaci jednoho důležitého typu tzv. lineárních datových struktur: seznamu. Pomocí něj pak můžeme reprezentovat rozličné abstraktní datové struktury, jako například množiny nebo polynomy. Nyní si ukážeme, jak lze páry a seznamy použít k reprezentaci tzv. **stromů**.

Nebudeme se pouštět do obecné definice stromu, ale ukážeme si nejjednodušší způsob, jak strom pomocí párů reprezentovat. *Strom* (*tree*) zadáme jako seznam *větví* (*branches*). Každá větev je buď opět strom, nebo *list* (*leaf*), což je cokoliv, co není seznam.

Takto si můžeme představit strom daný seznamem `(1 (2 3 (4 5)) (6 7))`:



Listy stromu nesou data, která strom uchovává. V našem příkladě jsou to hodnoty `1`, `2`, `3`, `4`, `5`, `6`, `7`. Každý list tedy má *hodnotu*.

Poučeni z minulých přednášek si teď napíšeme konstruktor a selektor pro stromy a listy:

```
(define (make-tree . branches)
  branches)

(define (tree-branches tree)
  tree)

(define (make-leaf value)
  value)

(define (leaf-value leaf)
  leaf)
```

Procedura `make-tree` vytvoří nový strom se zadanými větvemi, procedura `tree-branches` pro daný strom vrátí seznam jeho větví. Procedura `make-leaf` vytvoří list se zadanou hodnotou, procedura `leaf-value` vrátí hodnotu daného listu.

Procedury jsou triviální, protože triviální je reprezentace stromů a listů, ale je dobré si je na začátku napsat a dále se stromy a listy pracovat jako s abstraktními datovými strukturami (tohle jsme už podrobněji zdůvodňovali dříve). I když rozhodnutí v konkrétní situaci je vždy na programátorovi.

Také je dobré si uvědomit, že v této podobě jsou procedury napsány bez jakékoliv kontroly správnosti vstupu. Počítáme tedy s tím, že uživatel neudělá chybu. To by v realitě nebylo dostatečné. (Uživatel by mohl udělat chybu například v tom, že by jako argument procedury `make-leaf` použil seznam nebo že by jako argument procedury `tree-branches` nepoužil strom.)

Vytvoření stromu z obrázku pomocí konstruktorů:

```
> (make-tree (make-leaf 1)
             (make-tree (make-leaf 2)
                        (make-leaf 3)
                        (make-tree (make-leaf 4)
                                   (make-leaf 5)))
             (make-tree (make-leaf 6)
                        (make-leaf 7)))
(1 (2 3 (4 5)) (6 7))
```

Ze zadaného seznamu stromů nebo listů můžeme vytvořit nový strom pomocí procedury `apply`. Například strom, jehož větve máme uloženy v seznamu `branches`, vytvoříme takto: `(apply make-tree branches)`.

Typový predikát na list (budeme brzy potřebovat):

```
(define (leaf? x)
  (not (or (pair? x) (null? x))))
```

Při psaní procedur na práci se stromy si hezky procvičíme stromovou rekurzi, procedury vyššího řádu, práci se seznamy a vůbec. Následující procedura například zjistí počet listů stromu:

```
(define (leaf-count x)
  (if (leaf? x)
      1
      (apply + (map leaf-count (tree-branches x)))))
```

Součet hodnot listů stromu:

```
(define (tree-sum x)
  (if (leaf? x)
      (leaf-value x)
      (apply + (map tree-sum (tree-branches x)))))
```

Přičtení daného čísla k hodnotám všech listů stromu:

```
(define (tree-add tree-or-leaf val)
  (if (leaf? tree-or-leaf)
      (make-leaf (+ (leaf-value tree-or-leaf) val))
      (apply make-tree (map (lambda (x)
                              (tree-add x val))
                            (tree-branches tree-or-leaf)))))
```

A obecně, aplikace libovolné procedury na hodnoty všech listů stromu:

```
(define (tree-map proc tree-or-leaf)
  (if (leaf? tree-or-leaf)
      (make-leaf (proc (leaf-value tree-or-leaf)))
      (apply make-tree (map (lambda (x)
                              (tree-map proc x))
                            (tree-branches tree-or-leaf)))))
```

Procedura `tree-add` by nyní šla napsat takto:

```
(define (tree-add tree-or-leaf val)
  (tree-map (lambda (x) (+ x val)) tree-or-leaf))
```

Uvedené procedury jsou ovšem zbytečně složité. Můžeme je zjednodušit použitím pomocných procedur. Například u procedury `tree-map`:

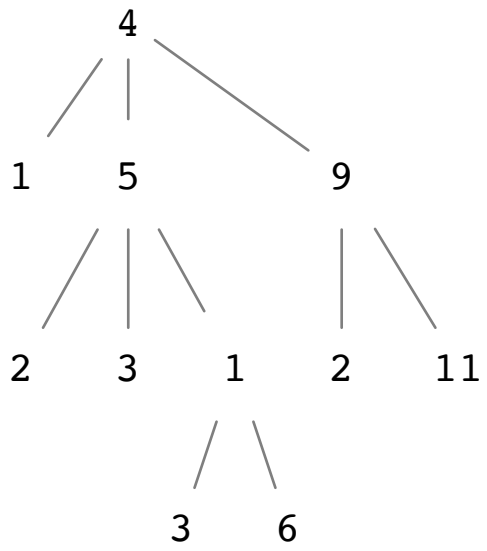
```
(define (tree-map proc tree-or-leaf)
  (if (leaf? tree-or-leaf)
      (tree-map-l proc tree-or-leaf)
      (tree-map-t proc tree-or-leaf)))

(define (tree-map-l proc leaf)
  (make-leaf (proc (leaf-value leaf))))

(define (tree-map-t proc tree)
  (apply make-tree (map (lambda (x)
                          (tree-map proc x))
                        (tree-branches tree))))
```

3 Stromy s hodnotami ve všech uzlech

Teď stromy vylepšíme tak, že hodnoty ponesou nejen v listech, ale i ostatních uzlech. Chceme tedy vytvořit vhodnou datovou reprezentaci pro takovéto stromy:



Uděláme to tak, že stromy budeme reprezentovat páry, jejichž *car* bude obsahovat hodnotu kořenového uzlu a *cdr* bude obsahovat seznam větví. Listy nyní můžeme chápat rovněž jako stromy, a to ty, které nemají žádnou větev. Strom na obrázku tedy bude reprezentován seznamem `(4 (1) (5 (2) (3) (1 (3) (6))) (9 (2) (11)))`.

Na tomto místě jsem bez pořádné definice začal používat pojem *uzel*. Věřím, že chápete, co mám na mysli.

Nyní bude zajímavé podívat se, jak je třeba přepsat už napsané procedury na práci se stromy. Některé tady s minimem komentářů uvedu.

Jelikož jsme změnili datovou reprezentaci stromů, je třeba změnit jejich konstruktory a selektory (a jeden selektor přidat):

```
(define (make-tree value . branches)
  (cons value branches))

(define (tree-value tree)
  (car tree))

(define (tree-branches tree)
  (cdr tree))
```

Listy jsou v nové reprezentaci jen speciálním typem stromu. Následující procedury by v principu nebylo nutné psát, ale kvůli úplnosti a zpětné kompatibilitě to uděláme:

```
(define (make-leaf value)
  (make-tree value))

(define (leaf-value leaf)
  (tree-value leaf))

(define (leaf? x)
  (null? (cdr x)))
```

Díky tomu, že jsme jak pro starou, tak pro novou verzi stromů napsali selektory a predikát na list, může procedura `leaf-count` zůstat beze změny:

```
(define (leaf-count x)
  (if (leaf? x)
      1
      (apply + (map leaf-count (tree-branches x)))))
```

Vidíme, jak je výhodné pracovat s abstraktními datovými strukturami čistě pomocí přístupových procedur (konstruktorů, selektorů, typových predikátů).

Procedura `tree-sum` bude mít jiný význam: sečte hodnoty nejen listů, ale i všech ostatních uzlů. Díky jednotnému chápání listů a stromů bude ovšem jednodušší než dříve:

```
(define (tree-sum x)
  (+ (tree-value x)
     (apply + (map tree-sum (tree-branches x)))))
```

Procedura `tree-map` by v nové verzi měla aplikovat zadanou proceduru na hodnoty všech uzlů (tedy nejen listů jako dříve). Bude rovněž jednodušší:

```
(define (tree-map proc tree)
  (apply make-tree
         (proc (tree-value tree))
         (map (lambda (x)
                (tree-map proc x))
              (tree-branches tree)))))
```

(Zde jsme použili obecnější možnost procedury `apply` a aplikovali jsme ji na tři argumenty. Podrobnosti najdete v dokumentaci k této proceduře.)

Otázky a úkoly na cvičení

První část úloh se týká jednoduchých stromů z části 2.

1. Vysvětlete, zda je vhodnější napsat predikát `leaf?` takto:

```
(define (leaf? x)
  (not (list? x)))
```

2. Procedura `leaf-count` vytváří zbytečně nové seznamy prostřednictvím procedury `map`. Upravte ji tak, aby to nedělala. Výhodné je použít na to proceduru `foldr`.
3. Udělejte totéž pro proceduru `tree-sum`.
4. Napište predikát `same-structure?`, který zjistí, zda dva stromy nebo listy mají stejnou strukturu, tj.
   (a) Musí to být dva stromy nebo dva listy.
   (b) Pokud jde o dva stromy, musí mít oba stejně dlouhé seznamy větví a odpovídající si větve musí mít opět stejnou strukturu.

Úlohy k části 3:

5. Bude vaše procedura `same-structure?` fungovat správně i na novou verzi stromů?
6. Procedura `tree-sum` je rekurzivní, ale zdá se, že neobsahuje ukončovací podmínku rekurze. Je to správně?

7. Napište proceduru `tree-to-list`, která k danému stromu vrátí seznam hodnot všech jeho uzlů. Prochází vaše procedura strom do hloubky, nebo do šířky? Pokud by jej procházela do hloubky, vrátila by pro strom z posledního obrázku například (je více možností, tato se nazývá *preorder*) seznam `(4 1 5 2 3 1 3 6 9 2 11)`. Průchodem do šířky bychom dostali `(4 1 5 9 2 3 1 2 11 3 6)`. Která z možností je jednodušší na naprogramování? Pokuste se napsat obě.